Python Dictionaries

A real-life dictionary holds words and their meanings. As you can imagine, likewise, a Python dictionary holds key-value pairs. Let's look at how to create one.

How to Create a Dictionary in Python?

Creating a Python Dictionary is easy as pie. Separate keys from values with a colon(:), and a pair from another by a comma(,). Finally, put it all in curly braces.

```
>>> {'PB&J':'Peanut Butter and Jelly','PJ':'Pajamas'}
```

Output

{'PB&J': 'Peanut Butter and Jelly', 'PJ': 'Pajamas'}

Optionally, you can put the dictionary in a variable. If you want to use it later in the program, you must do this.

```
>>> lingo={'PB&J':'Peanut Butter and Jelly','PJ':'Pajamas'}
```

To create an empty dictionary, simply use curly braces and then assign it to a variable

```
 >>> dict2={}
```

```
>>> dict2
```

Output

{}

```
>>> type(dict2)
```

Output

<class: dict>

1. **Python Dictionary Comprehension**

   You can also create a Python dict using comprehension. This is the same thing that you've learned in your math class. To do this, follow an expression pair with a for-statement loop in python. Finally, put it all in curly braces.

   ```
   >>> mydict={x*x:x for x in range(8)}
   ```

   ```
   >>> mydict
   ```

   Output

   {0: 0, 1: 1, 4: 2, 9: 3, 16: 4, 25: 5, 36: 6, 49: 7}

   In the above code, we created a Python dictionary to hold squares of numbers from 0 to 7 as keys, and numbers 0-1 as values.

2. Dictionaries with mixed keys

It isn't necessary to use the same kind of keys (or values) for a dictionary in Python.

```
>>> dict3={1:'carrots','two':[1,2,3]}
 >>> dict3
```

Output

{1: 'carrots', 'two': [1, 2, 3]} A

s you can see here, a key or a value can be anything from an integer to a list.

3. dict()

Using the dict() function, you can convert a compatible combination of constructs into a Python dictionary.

```
>>> dict(([1,2],[2,4],[3,6]))
```

Output

{1: 2, 2: 4, 3: 6}

However, this function takes only one argument. So if you pass it three lists, you must pass them inside a list or a tuple. Otherwise, it throws an error.

```
 >>> dict([1,2],[2,4],[3,6])
```

Output

Traceback (most recent call last):File "", line 1, in dict([1,2],[2,4],[3,6])

TypeError: dict expected at most 1 argument, got 3

4. Declaring one key more than once

Now, let's try declaring one key more than once and see what happens.

```
>>> mydict2={1:2,1:3,1:4,2:4}
>>> mydict2
```

Output

{1: 4, 2: 4}

As you can see, 1:2 was replaced by 1:3, which was then replaced by 1:4. This shows us that a dictionary cannot contain the same key twice.

5. Declaring an empty dictionary and adding elements later

When you don't know what key-value pairs go in your Python dictionary, you can just create an empty Python dict, and add pairs later.

```
>>> animals={}
>>> type(animals)
```

Output

<class: dict>

>>> animals[1]='dog'

>>> animals[2]='cat'

>>> animals[3]='ferret'

>>> animals

Output

{1: ‘dog’, 2: ‘cat’, 3: ‘ferret’}

How to Access a Python Dictionary?

1. Accessing the entire Python dictionary

To get the entire dictionary at once, type its name in the shell.

>>> dict3

Output

{1: ‘carrots’, ‘two’: [1, 2, 3]}

2. Accessing a value

To access an item from a list or a tuple, we use its index in square brackets. This is the python syntax to be followed. However, a Python dictionary is unordered. So to get a value from it, you need to put its key in square brackets. To get the square root of 36 from the above dictionary, we write the following code.

>>> mydict[36]

Output

6

3. get()

The Python dictionary get() function takes a key as an argument and returns the corresponding value.

>>> mydict.get(49)

Output

7

4. When the Python dictionary keys doesn’t exist

If the key you’re searching for doesn’t exist, let’s see what happens.

>>> mydict[48]

Output

Traceback (most recent call last):File “”, line 1, in mydict[48]

KeyError: 48 Using the key in square brackets gives us a KeyError.

Now let’s see what the get() method returns in such a situation.

>>> mydict.get(48)

>>>

As you can see, this didn’t print anything. Let’s put it in the print statement to find out what’s going on.

>>> print(mydict.get(48))

Output

None So we see, when the key doesn’t exist, the get() function returns the value None. We discussed it earlier, and know that it indicates an absence of value.

Reassigning a Python Dictionary

The python dictionary is mutable. This means that we can change it or add new items without having to reassign all of it.

1. Updating the Value of an Existing Key

If the key already exists in the Python dictionary, you can reassign its value using square brackets. Let’s take a new Python dictionary for this.

>>> dict4={1:1,2:2,3:3}

Now, let’s try updating the value for the key 2.

>>> dict4[2]=4

>>> dict4

Output

{1: 1, 2: 4, 3: 3}

2. Adding a new key

However, if the key doesn’t already exist in the dictionary, then it adds a new one.

>>> dict4[4]=6

>>> dict4

Output

{1: 1, 2: 4, 3: 3, 4: 6}

Python dictionary cannot be sliced.

How to Delete Python Dictionary?

You can delete an entire dictionary. Also, unlike a tuple, a Python dictionary is mutable. So you can also delete a part of it.

1. Deleting an entire Python dictionary

To delete the whole Python dict, simply use its name after the keyword 'del'.

```
>>> del dict4
 >>> dict4
```

Output

Traceback (most recent call last):File "", line 1, in dict4

NameError: name 'dict4' is not defined

2. Deleting a single key-value pair

To delete just one key-value pair, use the keyword 'del' with the key of the pair to delete. Now let's first reassign dict4 for this example.

```
>>> dict4={1:1,2:2,3:3,4:4}
```

Now, let's delete the pair 2:2

```
 >>> del dict4[2]
 >>> dict4
```

Output

{1: 1, 3: 3, 4: 4}

A few other methods allow us to delete an item from a dictionary in Python.

In-Built Functions on a Python Dictionary

A function is a procedure that can be applied on a construct to get a value. Furthermore, it doesn't modify the construct. Python gives us a few functions that we can apply on a Python dictionary. Take a look.

1. len()

The len() function returns the length of the dictionary in Python. Every key-value pair adds 1 to the length.

```
>>> len(dict4)
```

Output

3

An empty Python dictionary has a length of 0.

```
>>> len({})
```

2. any()

Like it is with lists an tuples, the any() function returns True if even one key in a dictionary has a Boolean value of True.

>>> any({False:False,":"})

Output

False

>>> any({True:False,"":""})

Output

True

>>> any({'1':",":"})

Output

True

3. all()

Unlike the any() function, all() returns True only if all the keys in the dictionary have a Boolean value of True.

>>> all({1:2,2:","":3})

Output

False

4. sorted()

Like it is with lists and tuples, the sorted() function returns a sorted sequence of the keys in the dictionary. The sorting is in ascending order, and doesn't modify the original Python dictionary. But to see its effect, let's first modify dict4.

>>> dict4={3:3,1:1,4:4} Now, let's apply the sorted() function on it.

>>> sorted(dict4)

Output

[1, 3, 4]

Now, let's try printing the dictionary dict4 again.

>>> dict4

Output

{3: 3, 1: 1, 4: 4}

As you can see, the original Python dictionary wasn't modified. This function returns the keys in a sorted list. To prove this, let's see what the type() function returns.

>>> type(sorted(dict4))

Output

<class: list>

This proves that sorted() returns a list.

In-Built Methods on a Python Dictionary

A method is a set of instructions to execute on a construct, and it may modify the construct. To do this, a method must be called on the construct. Now, let's look at the available methods for dictionaries.

Let's use dict4 for this example.

>>> dict4

Output

{3: 3, 1: 1, 4: 4}

1. keys()

The keys() method returns a list of keys in a Python dictionary.

>>> dict4.keys()

Output

dict_keys([3, 1, 4])

2. values()

Likewise, the values() method returns a list of values in the dictionary.

>>> dict4.values()

Output

dict_values([3, 1, 4])

3. items()

This method returns a list of key-value pairs.

>>> dict4.items()

Output

dict_items([(3, 3), (1, 1), (4, 4)])

4. get()

It takes one to two arguments. While the first is the key to search for, the second is the value to return if the key isn't found. The default value for this second argument is None.

>>> dict4.get(3,0)

Output

3

>>> dict4.get(5,0)

Since the key 5 wasn't in the dictionary, it returned 0, like we specified.

5. clear()

The clear function's purpose is obvious. It empties the Python dictionary.

>>> dict4.clear()

>>> dict4

Output

{}

Let's reassign it though for further examples.

>>> dict4={3:3,1:1,4:4}

6. copy()

First, let's see what shallow and deep copies are. A shallow copy only copies contents, and the new construct points to the same memory location. But a deep copy copies contents, and the new construct points to a different location. The copy() method creates a shallow copy of the Python dictionary.

>>> newdict=dict4.copy()

>>> newdict

Output

{3: 3, 1: 1, 4: 4}

7. pop()

This method is used to remove and display an item from the dictionary. It takes one to two arguments. The first is the key to be deleted, while the second is the value that's returned if the key isn't found.

>>> newdict.pop(4)

Output

4

Now, let's try deleting the pair 4:4.

>>> newdict.pop(5,-1)

Output

-1

However, unlike the get() method, this has no default None value for the second parameter.

>>> newdict.pop(5)

Output

Traceback (most recent call last):File “”, line 1, in newdict.pop(5)

KeyError: 5 As you can see, it raised a KeyError when it couldn’t find the key.

8. popitem()

Let’s first reassign the Python dictionary newdict.

>>> newdict={3:3,1:1,4:4,7:7,9:9}

Now, we’ll try calling popitem() on this.

>>> newdict.popitem()

Output

(9, 9)

It popped the pair 9:9.

>>> newdict={3:3,1:1,4:4,7:7,9:9}

>>> newdict.popitem()

Output

(9, 9)

As you can see, the same pair was popped. We can interpret from this that the internal logic of the popitem() method is such that for a particular dictionary, it always pops pairs in the same order.

9. fromkeys()

This method creates a new Python dictionary from an existing one. To take an example, we try to create a new dictionary from newdict. While the first argument takes a set of keys from the dictionary, the second takes a value to assign to all those keys. But the second argument is optional.

>>> newdict.fromkeys({1,2,3,4,7},0)

Output

{1: 0, 2: 0, 3: 0, 4: 0, 7: 0}

However, the keys don’t have to be in a set.

>>> newdict.fromkeys((1,2,3,4,7),0)

Output

{1: 0, 2: 0, 3: 0, 4: 0, 7: 0}

Like we've seen so far, the default value for the second argument is None.

>>> newdict.fromkeys({'1','2','3','4','7'})

Output

{'4': None, '7': None, '3': None, '1': None, '2': None}

10. update()

The update() method takes another dictionary as an argument. Then it updates the dictionary to hold values from the other dictionary that it doesn't already.

>>> dict1={1:1,2:2}

>>> dict2={2:2,3:3}

>>> dict1.update(dict2)

>>> dict1

Output

{1: 1, 2: 2, 3: 3}

Python Dictionary Operations We learned about different classes of **operators** in Python earlier. Let's now see which of those can we apply on dictionaries

1. Membership

We can apply the 'in' and 'not in' operators on a Python dictionary to check whether it contains a certain key. For this example, we'll work on dict1.

>>> dict1 Output {1: 1, 2: 2, 3: 3}

>>> 2 in dict1

Output

True

>>> 4 in dict1

Output

False

2 is a key in dict1, but 4 isn't. Hence, it returns True and False for them, correspondingly.

2. Python Iterate Dictionary

When in a for loop, you can also iterate on a Python dictionary like on a list, tuple, or set.

```
>>> dict4
```

Output

```
{1: 1, 3: 3, 4: 4}
```

```
>>> for i in dict4:
    print(dict4[i]*2)
```

Output

```
2
6
8
```

The above code, for every key in the Python dictionary, multiplies its value by 2, and then prints it.

3. Nested Dictionary

Finally, let's look at nested dictionaries. You can also place a Python dictionary as a value within a dictionary.

```
>>> dict1={4:{1:2,2:4},8:16}
>>> dict1
```

Output

```
{4: {1: 2, 2: 4}, 8: 16}
```

To get the value for the key 4, write the following code.

```
 >>> dict1[4]
```

Output

```
{1: 2, 2: 4}
```

However, you can't place it as a key, because that throws an error.

```
>>> dict1={{1:2,2:4}:8,8:16}
```

Output

```
Traceback (most recent call last):File "", line 1, in dict1={{1:2,2:4}:8,8:16}
TypeError: unhashable type: 'dict'
```

Example

#Program to build a dictionary of months for a year and prints the days in a month for the month name given as input

```
dom={"Jan":31,"Feb":28,"Mar":31,"Apr":30,"May":31,"Jun":30,"Jul":31,"Aug":31,"Sep":30,
"Oct":31,"Nov":30,"Dec":31}
print(dom)

while True:
        mon=input("Enter month")
        if mon in dom:
                print("No. of days in ",mon, " month are ", dom[mon])
        else:
                print(mon, " Month not found")
                ch=input("add one more [y/n]")
                if ch in ['y', 'Y']:
                        continue
                else:
                        break
```

Python Functions:

A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. A function can return data as a result.

Creating a Function

In Python a function is defined using the def keyword:

Example:

```
def my_function():
    print("Hello from a function")
```

Calling a Function

To call a function, use the function name followed by parenthesis:

Example:

```
def my_function():
    print("Hello from a function")
my_function()
```

Arguments

1. **Passing one argument:**

Information can be passed into functions as arguments. Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example:

```
def my_function(fname):
    print(fname + " Refsnes")
my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

Arguments *are often shortened to* args *in Python documentations.*

Parameters or Arguments?

The terms *parameter* and *argument* can be used for the same thing: information that are passed into a function.

From a function's perspective:

A parameter is the variable listed inside the parentheses in the function definition.

An argument is the value that is sent to the function when it is called.

2. Passing number of Arguments

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

Example

This function expects 2 arguments, and gets 2 arguments:

```
def my_function(fname, lname):
    print(fname + " " + lname)
my_function("Emil", "Refsnes")
```

If you try to call the function with 1 or 3 arguments, you will get an error:

Example:

This function expects 2 arguments, but gets only 1:

```
def my_function(fname, lname):
    print(fname + " " + lname)
my_function("Emil")
```

3. Arbitrary Arguments, *args

If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

This way the function will receive a *tuple* of arguments, and can access the items accordingly:

Example

If the number of arguments is unknown, add a * before the parameter name:

```
def my_function(*kids):
    print("The youngest child is " + kids[2])
my_function("Emil", "Tobias", "Linus")
```

Arbitrary Arguments are often shortened to **args* in Python documentations.

4. Keyword Arguments

You can also send arguments with the *key* = *value* syntax.

This way the order of the arguments does not matter.

Example

```
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)
my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

The phrase *Keyword Arguments* are often shortened to *kwargs* in Python documentations.

5. Arbitrary Keyword Arguments, **kwargs

If you do not know how many keyword arguments that will be passed into your function, add two asterisk: ** before the parameter name in the function definition.

This way the function will receive a *dictionary* of arguments, and can access the items accordingly:

Example

If the number of keyword arguments is unknown, add a double ** before the parameter name:

```
def my_function(**kid):
  print("His last name is " + kid["lname"])
my_function(fname = "Tobias", lname = "Refsnes")
```

Arbitrary Kword Arguments are often shortened to ***kwargs* in Python documentations.

6. Default Parameter Value

The following example shows how to use a default parameter value.

If we call the function without argument, it uses the default value:

Example

```
def my_function(country = "Norway"):
  print("I am from " + country)
my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

7. Passing a List as an Argument

You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

E.g. if you send a List as an argument, it will still be a List when it reaches the function:

Example

```
def my_function(food):
    for x in food:
        print(x)
fruits = ["apple", "banana", "cherry"]
my_function(fruits)
```

Return Values

To let a function return a value, use the return statement:

Example

```
def my_function(x):
  return 5 * x
print(my_function(3))
print(my_function(5))
print(my_function(9))
```

The pass Statement

function definitions cannot be empty, but if you for some reason have a function definition with no content, put in the pass statement to avoid getting an error.

Example

```
def myfunction():
  pass
```

Example:

```
def add(a,b):
      return a+b
c=add(10,15)
print("sum is",c)
```

output:

sum is 25

Example programs to practice:

1. **Addition of matrices by using functions**
2. **Factorial**
3. **Fibonacci**
4. **Linear search**
5. **Binary search**
6. **Selection sort**
7. **Bubble sort**

All programs are uploaded to given drive link

Recursion

Python also accepts function recursion, which means a defined function can call itself without using loops.

Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

Example 1:

```
def fact(num):
    if(num==1):
        return 1
    else:
        return (num*fact(num-1))
num=int(input("enter a number"))
i=1
print(fact(num))
```

Output:

```
Enter a number 5
120
```

Example 2:

```
def fib(num):
    if(num==0):
        return 0
    elif(num==1):
```

```
        return 1
    else:
        return(fib(num-2)+fib(num-1))
num=int(input("\n enter a number"))
for num in range(0,num):
    print(fib(num))
```

Output:

Enter a number 6

0 1 1 2 3 5

Python Mathematical Functions

The **math** module is used to access mathematical functions in the Python. All methods of this functions are used for integer or real type objects, not for complex numbers.

To use this module, we should import that module into our code.

```
import math
```

Some Constants

These constants are used to put them into our calculations.

S.No.	Constants & Description
1	**pi** Return the value of pi: 3.141592
2	**E** Return the value of natural base e. e is 0.718282
3	**tau** Returns the value of tau. tau = 6.283185
4	**inf** Returns the infinite
5	**nan** Not a number type.

Numbers and Numeric Representation

These functions are used to represent numbers in different forms. The methods are like below –

S.No.	Function & Description
1	**ceil(x)** Return the Ceiling value. It is the smallest integer, greater or equal to the number x.
2	**copysign(x, y)** It returns the number x and copy the sign of y to x.
3	**fabs(x)** Returns the absolute value of x.
4	**factorial(x)** Returns factorial of x. where $x \geq 0$
5	**floor(x)** Return the Floor value. It is the largest integer, less or equal to the number x.
6	**fsum(iterable)** Find sum of the elements in an iterable object
7	**gcd(x, y)** Returns the Greatest Common Divisor of x and y
8	**isfinite(x)** Checks whether x is neither an infinity nor nan.
9	**isinf(x)** Checks whether x is infinity
10	**isnan(x)** Checks whether x is not a number.
11	**remainder(x, y)** Find remainder after dividing x by y.

Example Code

```
import math
print('The Floor and Ceiling value of 23.56 are: ' + str(math.ceil(23.56)) + ', ' +
str(math.floor(23.56)))
x = 10
y = -15
print('The value of x after copying the sign from y is: ' + str(math.copysign(x, y)))
print('Absolute value of -96 and 56 are: ' + str(math.fabs(-96)) + ', ' + str(math.fabs(56)))
my_list = [12, 4.25, 89, 3.02, -65.23, -7.2, 6.3]
print('Sum of the elements of the list: ' + str(math.fsum(my_list)))
print('The GCD of 24 and 56 : ' + str(math.gcd(24, 56)))
x = float('nan')
if math.isnan(x):
   print('It is not a number')
x = float('inf')
y = 45
if math.isinf(x):
   print('It is Infinity')
print(math.isfinite(x)) #x is not a finite number
print(math.isfinite(y)) #y is a finite number
```

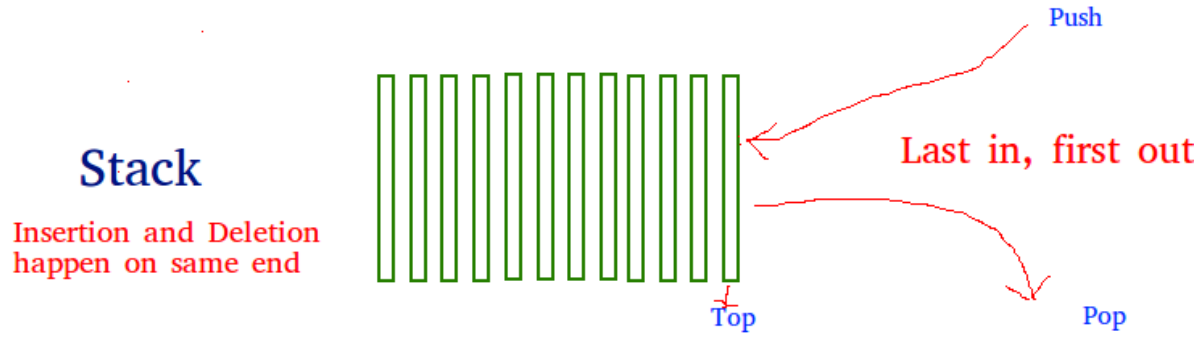
Output

```
The Floor and Ceiling value of 23.56 are: 24, 23
The value of x after copying the sign from y is: -10.0
Absolute value of -96 and 56 are: 96.0, 56.0
Sum of the elements of the list: 42.13999999999999
The GCD of 24 and 56 : 8
It is not a number
It is Infinity
False
True
```

User Defined Data Types:

Stack

A **stack** is a linear data structure that stores items in a **Last-In/First-Out (LIFO)** or First-In/Last-Out (FILO) manner. In stack, a new element is added at one end and an element is removed from that end only. The insert and delete operations are often called push and pop.



The functions associated with stack are:

- **empty()** – Returns whether the stack is empty – Time Complexity: O(1)
- **size()** – Returns the size of the stack – Time Complexity: O(1)
- **top() / peek()** – Returns a reference to the topmost element of the stack – Time Complexity: O(1)
- **push(a)** – Inserts the element 'a' at the top of the stack – Time Complexity: O(1)
- **pop()** – Deletes the topmost element of the stack – Time Complexity: O(1)

Implementation of stack using list

```
stack = []
stack.append('a')
stack.append('b')
stack.append('c')
print('Initial stack')
print(stack)
print('\nElements popped from stack:')
print(stack.pop())
print(stack.pop())
print(stack.pop())
print('\nStack after elements are popped:')
print(stack)
```

Output

Initial stack

['a', 'b', 'c']

Elements popped from stack:

c

b

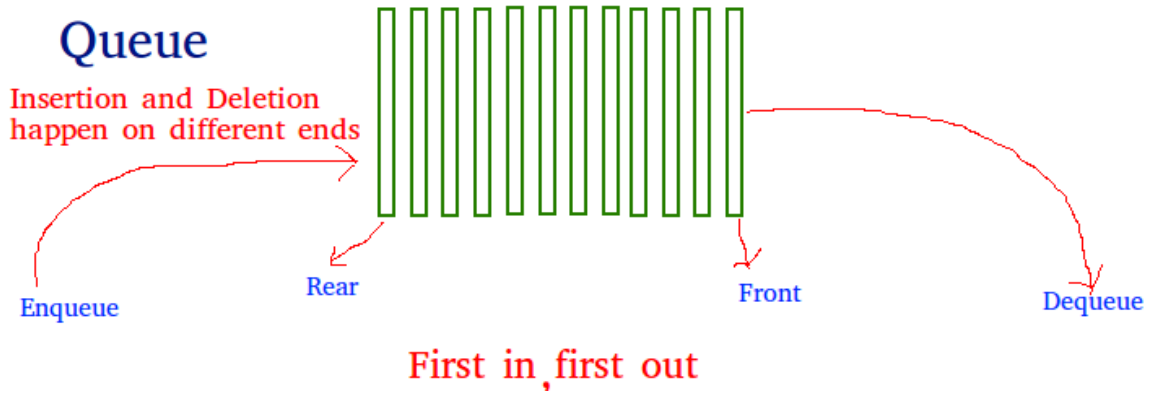
a

Stack after elements are popped:

[]

Queue in Python

Like stack, queue is a linear data structure that stores items in First In First Out (FIFO) manner. With a queue the least recently added item is removed first. A good example of queue is any queue of consumers for a resource where the consumer that came first is served first.



Operations associated with queue are:

- **Enqueue:** Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition – Time Complexity : O(1)
- **Dequeue:** Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition – Time Complexity : O(1)
- **Front:** Get the front item from queue – Time Complexity : O(1)
- **Rear:** Get the last item from queue – Time Complexity : O(1)

Implementation using list

```
queue = []
queue.append('a')
queue.append('b')
queue.append('c')
print("Initial queue")
print(queue)
print("\nElements dequeued from queue")
print(queue.pop(0))
print(queue.pop(0))
print(queue.pop(0))
print("\nQueue after removing elements")
print(queue)
```

Output:

Initial queue

['a', 'b', 'c']

Elements dequeued from queue

a

b

c

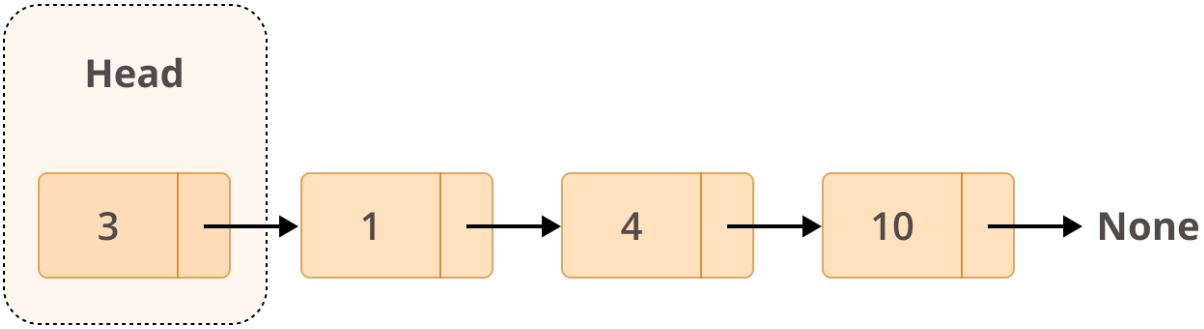
Queue after removing elements

[]

Linked List

Linked list is a simple data structure in programming, which obviously is used to store data and retrieve it accordingly. To make it easier to imagine, it is more like a dynamic array in which data elements are linked via pointers (i.e. the present record points to its next record and the next one points to the record that comes after it, this goes on until the end of the structure) rather than being tightly packed.
There are two types of linked list:

1. Single-Linked List: In this, the nodes point to the node immediately after it



2. Doubly Linked List: In this, the nodes not only reference the node next to it but also the node before it.